Trace-Based Control-Flow Analysis
Benoît Montagu, Thomas Jensen

To cite this version:
Benoît Montagu, Thomas Jensen. Trace-Based Control-Flow Analysis. PLDI 2021 - 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Jun 2021, Virtual, Canada. pp.1-15, 10.1145/3453483.3454057. hal-03266981

HAL Id: hal-03266981
https://inria.hal.science/hal-03266981
Submitted on 22 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Trace-Based Control-Flow Analysis

Benoît Montagu
Inria
Rennes, France
benoit.montagu@inria.fr

Thomas Jensen
Inria
Rennes, France
thomas.jensen@inria.fr

Abstract
We define a small-step semantics for the untyped \(\lambda\)-calculus, that traces the \(\beta\)-reductions that occur during evaluation. By abstracting the computation traces, we reconstruct \(k\)-CFA using abstract interpretation, and justify constraint-based \(k\)-CFA in a semantic way. The abstract interpretation of the trace semantics also paves the way for introducing widening operators in CFA that go beyond existing analyses, that are all based on exploring a finite state space. We define VCFA, a widening-based analysis that limits the cycles in call stacks, and can achieve better precision than \(k\)-CFA at a similar cost.

CCS Concepts:
- Software and its engineering → Automated static analysis;
- Theory of computation → Operational semantics; Program analysis; Abstraction; Functional constructs.

Keywords: \(\lambda\)-calculus, control flow analysis, program traces, abstract interpretation, widening

ACM Reference Format:

1 Introduction
Control-flow analysis (CFA) [25, 42] is a family of static analyses whose goal is to determine which functions can be called at a given program point. For this purpose, CFAs generally compute two maps: a cache that determines the closures that each program point may evaluate to, and an environment that describes the values that could be assigned to each variables. With this information, it becomes possible to inline functions, and to remove safely some dynamic tag checks that must be performed in dynamically-typed languages. CFA of higher-order or object-oriented languages is difficult, since the control flow graph of a program is not static. Thus, CFAs try to compute over-approximations of the dynamic call graph. In the functional language community, two main research tracks on CFAs have emerged.

A first group of work has employed constraints to model the conditions under which a solution to a CFA problem is correct. Work in this vein [13, 16, 31, 33, 46] transforms a given program into a logical constraint—usually set-based constraints—that must be solved by a second program—the constraint solver. We refer to this research as “constraint-based analysis” (CBA).

A second group of work has focused on analysing abstract machines, and devised abstract interpretations of (small-step) abstract-machine-based semantics [15, 18, 28, 28]. In this “abstracting abstract machines” (AAM) research track, the low-level elements of abstract machines—environments, stores, continuation stacks—must be abstracted. A related approach has been applied to definitional (big-step) interpreters [4, 9].

The methodology of abstract interpretation [3] has been successfully applied to the AAM approach to justify the soundness of AAM-based analyses. The CBA approach, on the other hand, has focused on syntactic techniques, such as preservation properties, similar to the syntactic type soundness technique. More semantic justifications for the CBA approach, based on abstract interpretation, are still missing.

Although they differ in many ways, the AAM and CBA families have something in common: to ensure the termination of their analyses, they both perform approximations so that their search space becomes finite. Based on the finiteness of the search space, they are able to prove complexity results for CFAs [43]. This finiteness, however, impedes the use of abstract domains with unbounded heights. One such domain is the interval domain, that abstracts sets of integers. The restriction to a finite search space has also diminished the attention to widening techniques [3] for CFAs. While widening is often necessary to ensure the termination of analysers, it is also beneficial in finite domains, where it accelerates the convergence to a solution.

In this paper, we seek to reconcile CFA with abstract interpretation in two ways. First, we show that the control-flow analysis of a \(\lambda\)-term is an abstraction of its execution trace. This trace records two kinds of events: which function calls occurred, and which values were returned at a program point. This suffices to reconstruct the caches and environments found in CFAs. We designed the semantics so that the events track the contexts in which they are produced (at which program point, and after which function call). We
show how to recover the usual notion of call strings [40] from this contextual information. This provides a formalisation of the folklore knowledge that “context-sensitivity is trace partitioning”. This process of building traces and abstracting them is general [2] and can be applied with benefits to other static analyses. Based on this abstraction, we give a modular soundness proof of the constraint-based approach.

In a second stage, we derive by abstract interpretation the \( \nabla \text{CFA} \) analysis, an expressive control-flow analysis that is built on an infinite domain, and that employs widening to ensure convergence. The precision of this analysis indicates that further research on widening techniques for CFAs is worthwhile.

The contributions of the paper are organised as follows:

- We define a simple, substitution-based small-step semantics for the \( \lambda \)-calculus, that tracks the values produced in each evaluation contexts and the \( \beta \)-reductions that are performed during reduction (§2).

- We define a stack of abstractions (§3) from which we build the global environment and cache found in standard CFAs [33] out of the computation trace of a program. As a methodology, we derive for each abstraction layer the constraints that are satisfied by the abstracted semantics. At the end of our abstraction stack, we obtain a family of CFAs that generalise \( k \)-CFA, and that is parameterised by context sensitivity policies.

- Using the previous abstractions, we define a semantic criterion for the solutions of a constraint-based CFA problem. Using this criterion, we give a novel, modular proof of soundness for the constraint rules of \( k \)-CFA (§4), where each rule is verified separately. This gives a semantic justification for constraint-based CFA.

- Based on our semantic development, we present \( \nabla \text{CFA} \) (§5), a novel control-flow analysis that goes beyond the finite state abstraction used in traditional CFAs. \( \nabla \text{CFA} \) employs a widening operator, that is used at specific points, that are determined by the \( k^* \) heuristic by limiting call strings to “at most \( k \) cyclic call sites”. It can replace the “last \( k \) call sites” heuristic from \( k \)-CFA.

- Based on the implementation of the analysis, we discuss experimental results (§6), by comparing 0-CFA and 1-CFA with our new analysis and heuristics on a set of example programs taken from the literature. The new analysis takes advantage of non-finite integer domains, such as intervals, to compute more precise approximations with a similar computational cost.

2 A Labelled \( \lambda \)-Calculus

2.1 Syntax

We assume the existence of three infinite sets \( \mathcal{X}, \mathcal{L} \) and \( \mathbb{P} \). The set of variables \( \mathcal{X} \) contains variables, that are denoted by the metavariables \( x, y, z \). The set \( \mathcal{L} \) contains labels, that are denoted by the metavariable \( \ell \). Labels are put on lambda abstractions and serve to identify binding sites of variables. The set \( \mathbb{P} \) contains program points (also called locations), that are denoted by the metavariable \( p \).

**Definition 2.1** (Syntax). The set of terms \( \mathcal{T} \) is inductively defined as follows:

\[
t \in \mathcal{T} ::= x \quad \text{(Variables)}
\]
\[
\ell x. t \quad \text{(Abstraction)}
\]
\[
t t \quad \text{(Application)}
\]
\[
[t]^a \quad \text{(Annotation)}
\]

The terms of the language are the terms of the untyped \( \lambda \)-calculus extended with the annotation construct \( [t]^a \), that has no computational meaning, but is used to collect information about the evaluation of \( t \) using the annotation \( a \).

We introduce two kinds of annotations: program points \( p \in \mathbb{P} \) and function calls Call\( (\ell x. t, v) \). Program point annotations will serve to record (syntactic) call sites, and function call annotations to keep track of the terms actually applied in the call. Annotation paths \( \pi \) are sequences of annotations, and will serve as contexts. For example, a term \( [(\ell x. t) v]^p \) reduces to \( [[[(x \leftarrow v)]^e]^\text{Call}(\ell x. t, v)]^\text{Call}(\ell x. t, v)]^p \), and any subsequent reduction will happen in a context of the form \( \pi = p \cdot \text{Call}(\ell x. t, v) \cdot \pi' \). This context tells us that the call site of the actual call is \( p \). We let \( \delta \) range over call site paths i.e. those annotation paths that only consist of program points. We write \( \pi_1 \cdot \pi_2 \) for the concatenation of paths.

**Definition 2.2** (Annotations and paths).

\[
a \equiv p \quad | \quad \text{Call}(\ell x. t, v) \quad \text{(Annotations)}
\]
\[
\pi \equiv \varepsilon \quad | \quad \pi a \quad \text{(Annotation paths)}
\]
\[
\delta \in \mathcal{D} \equiv \varepsilon \quad | \quad \delta p \quad \text{(Call site paths)}
\]

The set of free variables \( \text{fv} t \) of a term \( t \) follows the standard definition. We write \( t[x \leftarrow u] \) for the term \( t \) where the variable \( x \) is replaced by the term \( u \) (without capture of variables). We omit its definition. The set of values \( \mathbb{V} \) is standard for a call by value \( \lambda \)-calculus.

**Definition 2.3** (Values). \( v \in \mathbb{V} ::= \ell x. t \).
(possibly empty) sequences of events. We write \( \text{tr}_1 \uplus \text{tr}_2 \) for the concatenation of traces.

**Definition 2.4** (Events and traces).
\[
e \equiv \text{Ret}(\pi, a, v) \mid \beta(\pi, \text{x} \cdot t, v) \quad \text{(Events)}
\]
\[
\text{tr} \in \mathcal{T} \equiv \varepsilon \mid e, \text{tr} \quad \text{(Traces)}
\]

The operational semantics uses the following operator for prefixing all events in a trace with an annotation path \( \pi \).

**Definition 2.5** (Prefixing by a path).
\[
\pi \cdot \text{Ret}(\pi', a, v) \equiv \text{Ret}(\pi \cdot \pi', a, v)
\]
\[
\pi \cdot \beta(\pi', \text{x} \cdot t, v) \equiv \beta(\pi \cdot \pi', \text{x} \cdot t, v)
\]
\[
\pi \cdot \varepsilon \equiv \varepsilon
\]
\[
\pi \cdot (e, \text{tr}) \equiv (\pi \cdot e), (\pi \cdot \text{tr})
\]

The notation \( t \xrightarrow{e} u \) denotes that the term \( t \) reduces in one step to \( u \), and during this reduction, produces the event \( e \). The definition of the one-step reduction is:
\[
(\text{x} \cdot \text{tr}) 
\begin{array}{c}
\beta(\text{x} \cdot \text{tr}, \text{x} \cdot t, v) \\
\text{Ret}(\text{c}, \text{a}, v)
\end{array} 
\xrightarrow{[t] \xrightarrow{e} [u]^{\text{Call}(\text{x} \cdot \text{tr}, \text{x} \cdot t, v)}} [v]^{\text{Ret}(\text{c}, \text{a}, v)}
\]

As we mentioned earlier, the contraction of a \( \beta_{\text{x}} \) redex performs a substitution and encloses the reduction in an annotation, so as to remember that the reductions that will follow happen in the context of a function call. This is indeed what is meant by the contextual rule for annotations: any event that is emitted under an annotation is reported by prefixing that event with the annotation. Thus, annotations record the contexts of reductions.

The reflexive transitive closure is written \( t \rightarrow^* u \), and defined as follows:
\[
t \xrightarrow{e} t \rightarrow t_1 \xrightarrow{e} t_2 \rightarrow^* t_3 \xrightarrow{e} t_3
\]
\[
t \xrightarrow{e} t_3
\]

The rule for reflexivity traces no event, and the rule for transitivity collects the events in the order they occur.

**Lemma 2.6.** The following inference rules are valid:
\[
t \xrightarrow{e} t_1 \xrightarrow{e} t_2 \xrightarrow{e} t_3
\]
\[
t \xrightarrow{e} t \xrightarrow{e} t_3
\]
\[
[t] \xrightarrow{e} [u]^{\text{Ret}(\text{c}, \text{a}, v)}
\]

We give an example of term reduction in Fig. 1. The first step in the evaluation reduces the term annotated with 1 which returns a function. Similarly, the second step evaluates the term annotated with 5 to produce the argument to the function. The third step is the \( \beta \)-reduction that applies the function to the argument, emitting a \( \beta \)-event which states that a \( \beta \)-reduction occurred in the context 0. The two following return steps illustrate how the same lambda term \( \lambda_0 \cdot y \cdot [y]^0 \) can be returned under different annotations (here, \( \{ [\lambda_0 \cdot y \cdot [y]^0] \xrightarrow{[\lambda_0 \cdot y \cdot [y]^0]} [\lambda_0 \cdot y \cdot [y]^0] \xrightarrow{[\lambda_0 \cdot y \cdot [y]^0]} \}

**Figure 1.** Example of reductions in the labelled \( \lambda \)-calculus.

3 and 4), thereby distinguishing when it is used as a function and as an argument. The context in those two return events is the path \( 0 \cdot C_1 \cdot 2 \) which contains the information that the returned value was obtained by first evaluating the call site annotated with 0, thereby initiating the call \( C_1 \), which in turn led to the call site labelled 2.

### 2.3 Trace Collecting Semantics

We use \( \sigma \) to range over value substitutions, i.e., finite mappings from variables to closed values. The expression \( t \cdot \sigma \) denotes the application of the value substitution \( \sigma \) to the term \( t \); it amounts to replacing every free occurrence of \( x \) in \( t \) with \( \sigma(x) \). We write \( \sigma[x \mapsto v] \) for the substitution \( \sigma \) extended with a new binding that maps \( x \) to \( v \). Given a set of substitutions \( I \), we write \( I \cdot \{x \mapsto v\} \) for the set \( \{\sigma[x \mapsto v] \mid \sigma \in I\} \), and write \( I \cdot \{x \mapsto S\} \) for the set \( \{\sigma[x \mapsto v] \mid v \in S\} \).

We consider the following semantics for terms that, given a set of input substitutions, produces a set of pairs, composed of the execution trace and the output value.

**Definition 2.7.** \( \{[t]_I \mid t \cdot \sigma \xrightarrow{e} v, \sigma \in I\} \)

**Definition 2.8** (Semantic preorder). We write \( \text{tr}_1 \preceq \text{tr}_2 \) when \( \text{tr}_1 \) is a sub-sequence of \( \text{tr}_2 \). It is an order relation. We use the same symbol to denote the following pre-order on sets of traces and values:

\[
S_1 \preceq S_2 \equiv \forall (t_1, v) \in S_1, \exists t_2, \text{tr}_1 \preceq \text{tr}_2 \land (t_2, v) \in S_2.
\]

The pre-order \( \preceq \) will serve as the approximation ordering. Larger sets of traces and values are less precise, in the sense that they might contain more traces and output values, and might also contain traces with more events.
Program reduction and trace ordering are related by the following lemma:

**Lemma 2.9 (Preservation).** If \( t_1 \xrightarrow{\text{tr}} t_2 \), then \( \langle t_2 \rangle_I \preceq \langle t_1 \rangle_I \).

The above lemma is central for our semantic framework: it implies that any correct over-approximation of the semantics of \( t_1 \) (in the sense of \( \preceq \)) is necessarily a sound approximation of any reduct of \( t_1 \). In other words, the semantic property of being a sound approximation is preserved by reduction. This fact is crucial for the preservation property of CFA constraints from §4 (Lemma 4.2). The proof of Lemma 2.9 is direct: the output values of \( t_1 \) and \( t_2 \) are the same, and the trace produced by \( t_2 \) is a suffix—and thus a sub-sequence—of the trace of \( t_1 \).

In the rest of this paper, we will only consider **fully annotated** terms as our source language, i.e., terms where variables, applications and abstractions are enclosed in exactly one annotation, which must be a program point. A fully annotated value is an abstraction of a fully annotated term.

For each abstraction that we define in the rest of the paper, we derive the inclusion properties that are satisfied by the abstract semantics. These inclusions naturally lead to the design of analysers in the style of big-step static interpreters [4, 9], that follow the structure of programs.

For fully annotated terms, we get the following inclusions for the trace semantics:

**Lemma 2.10 (Semantic inclusions for annotated terms).**

\[
\begin{align*}
\langle x \rangle^p_I & = \{(\text{Ret}(c, p, \sigma(x)), \sigma(x)) \mid \sigma \in I\} \\
\langle \lambda x. t \rangle^p_I & = \{(\text{Ret}(c, p, (\lambda x. t) \cdot \sigma), (\lambda x. t) \cdot \sigma) \mid \sigma \in I\} \\
\langle t_1 \rangle^p_{1} \cdot \langle t_2 \rangle^p_{2} \rangle_I & \subseteq \\
\langle \beta(p, \lambda x. t_1) \rangle^p_{1} \cdot \langle \beta(p, \lambda x. t_2) \rangle^p_{2} \rangle_I & \subseteq \\
\langle \text{Ret}(p, \text{Call}(\lambda x. t_1)) \rangle^p_{1} \cdot \langle \text{Ret}(p, \text{Call}(\lambda x. t_2)) \rangle^p_{2} \rangle_I & \subseteq
\end{align*}
\]

For variables, the semantics produces return events only, where the returned values are read in the possible input substitutions. The case for functions is similar: the returned values are functions whose free variables are substituted for the values given by the input variables.

In the inclusion for fully annotated applications, the execution trace \( t_3 \) that corresponds to the function call is **enclosed** in the following two events: the event \( \beta(p, \lambda x. t_1) \) that denotes the **start** of the function call, and the event \( \text{Ret}(p, \text{Call}(\lambda x. t_1)) \), that denotes the **return** from that function call. Interestingly, the same trace \( t_3 \) is **prefixed** with the location \( p \cdot \text{Call}(\lambda x. t_1) \), that records which function call is considered, and where it happened. In usual CFA analyses for functional languages, only the **locus** \( p \) of the function call is remembered.

\[
\begin{align*}
\mathcal{T} & \rightarrow \mathcal{R}(\mathcal{Clos}) \times (X^\text{fin} \rightarrow (L \times D)) \rightarrow \mathcal{R}(\mathcal{Clos}) \times C(\mathcal{Clos}) \\
\text{Y} & \rightarrow \alpha \text{context projection (§1.7)} \\
\mathcal{T} & \rightarrow \mathcal{R}(\mathcal{Clos}) \times (X^\text{fin} \rightarrow (L \times D)) \rightarrow \mathcal{R}(\mathcal{Clos}) \times C(\mathcal{Clos}) \\
\mathcal{Y}_\text{Clos} & \rightarrow \text{induction abstraction (§4.3)} \\
\mathcal{T} & \rightarrow (X^\text{fin} \rightarrow \varphi(\mathcal{Clos})) \rightarrow \mathcal{R}(\mathcal{Clos}) \times C(\mathcal{Clos}) \\
\mathcal{Y}_{\text{clo}} & \rightarrow \text{closure abstraction (§3.5)} \\
\mathcal{T} & \rightarrow (X^\text{fin} \rightarrow \varphi) \rightarrow \mathcal{R}(\mathcal{V}) \times C(\mathcal{V}) \\
\mathcal{Y}_\text{M} & \rightarrow \alpha \text{ independent attribute for inputs (§1.4)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \mathcal{R}(\mathcal{V}) \times C(\mathcal{V}) \\
\mathcal{Y}_{\text{EC}} & \rightarrow \alpha \text{ independent attribute for maps (§1.3)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \varphi(\mathcal{R}(\mathcal{V}) \times C(\mathcal{V})) \\
\mathcal{Y}_{\text{LA}} & \rightarrow \alpha \text{ independent attribute for pairs (§1.3)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \varphi(\mathcal{R}(\mathcal{V}) \times C(\mathcal{V}) \times \mathcal{V}) \\
\mathcal{Y}_{\text{NoVal}} & \rightarrow \alpha \text{ output oblivious (§1.3)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \varphi(\mathcal{R}(\mathcal{V}) \times C(\mathcal{V}) \times \mathcal{V}) \\
\mathcal{Y}_h & \rightarrow \alpha \text{ history oblivious (§1.2)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \varphi(\mathcal{T} \times \mathcal{V}) \\
\mathcal{Y}_{\text{tc}} & \rightarrow \alpha \text{ event selection (§1.1)} \\
\mathcal{T} & \rightarrow \varphi(X^\text{fin} \rightarrow \mathcal{V}) \rightarrow \varphi(\mathcal{T} \times \mathcal{V})
\end{align*}
\]

**Figure 2.** Abstraction steps.

### 3 A Systematic Reconstruction of CFA

Standard control flow analyses for a program \( t \) construct two objects as results of the analysis. The first object is a **global environment** \( \hat{\rho} \) that records the set of values that variables of \( t \) could have been assigned to. To distinguish between several instances of variables, variables are distinguished by their calling context—sometimes also called memento [31, 33] or instances [37]. A global environment maps a pair \((x, \delta)\) of a variable and its calling context to a set of abstract values. The global environment is called **global store** in the AAM line of work. In this paper, instead of taking variables and contexts \((x, \delta)\) as keys of global environments, we choose **binding sites** (corresponding to \(\lambda\)-abstractions) and contexts \((\ell, \delta)\) as keys. This choice is technical and can be ignored in a first read: it ensures that our definitions do not depend on the names of variables, and are consistent with the \(\alpha\)-equivalence of \(\lambda\)-terms. We write \( \mathbb{R}(A) \xrightarrow{\alpha} (L \times D) \rightarrow \varphi(A) \) for the type of global environments that contain sets of elements of type \( A \).

The second object computed by a CFA is a **global cache** \( \hat{C} \), that maps every sub-term of the program \( t \)—identified by their program points—to a set of abstract values that denotes the possible evaluation results of that sub-term. Similarly
to global environments, different versions of sub-terms are distinguished by their evaluation contexts. A global cache therefore maps a pair \((p, \delta)\) of a program point and an evaluation context to a set of abstract values. Some analyses remove the need of a cache by working with administrative normal forms (ANF), where every sub-term is bound to a variable. We write \(\mathbb{C}(A) \equiv (\mathbb{P} \times \mathbb{D}) \to \mathcal{P}(A)\) for the type of global caches that contain sets of elements of type \(A\).

In this section, we show how to reconstruct the environments and caches of \(k\)-CFA as abstractions of the trace-based semantics. The abstraction strategy is summarised in Fig. 2. Most abstractions form Galois connections \([3]\). A Galois connection between partially ordered sets is defined as follows:

**Definition 3.1** (Galois connection). We write \((A, \leq_A) \leq_{T, A} (B, \leq_B)\) when for any \(a \in A\) and \(b \in B\), \(a \leq_A b \iff a \leq_A \gamma(b)\).

The first stages of abstraction—starting from the bottom of Fig. 2—are responsible for selecting events in the traces, and forgetting about the ordering of events. After these preliminary steps, we have built environments and caches of types \(\mathbb{R}(V)\) and \(\mathbb{C}(V)\), that contain sets of values. Then, inputs are abstracted and finally values are abstracted by the closure abstraction and the indirection abstraction. These two abstractions are not Galois connections, as there is no best way to perform these abstractions, but they still lead to correct analyses.

The closure abstraction builds abstract closures, composed of some code and some environment, that maps variables to sets of abstract closures. Then, the indirection abstraction introduces intermediate names of type \(\mathbb{L} \times \mathbb{D}\): at this level, abstract closures are composed of some code together with an environment that maps variables to names. This indirection is similar to the indirection used in the small-step semantics of the AAM line of work, where names are called *addresses*.

Finally, the last abstraction step modifies the calling contexts in names using a projection function \(\phi\): we recover \(k\)-CFA by choosing \(\phi(\delta) = [\delta]_k\), that is the truncation to the last \(k\) elements of \(\delta\).

In the following subsections, we define the abstractions of the semantics and, at some key steps, show how they impact the semantic inclusions. We exploit these inclusions in §4 to semantically justify the constraints of \(k\)-CFA.

### 3.1 Abstraction of Events

We select events by performing filtering on traces. First, we select the *call sites* in the contexts of events. The call sites are the program points that occur just before a \(\text{Call}(v_1, v_2)\) annotation. Recall the example from §2.1 with \([[\lambda x. t] v]^p\) that reduces to \([[t[x \leftarrow v]]^\text{Call}(\lambda x. t, v)]]^p\). Subsequent reductions will happen in a context of the form \(\pi = p \cdot \text{Call}(\lambda x. t, v) \cdot \pi'\), recording that the call site of the actual call is \(p\). Thus, to recover the standard call site information we just need to keep the information \(\text{calls}(\pi) = p \cdot \text{calls}(\pi')\).

The call filtering is extended to events and traces as follows:

**Definition 3.2** (Call filtering).

- On paths: \(\text{calls}(\epsilon) \triangleq \text{calls}((\cdot \cdot) \cdot \epsilon) \triangleq \epsilon\)
  \(\text{calls}(p \cdot \text{Call}(v_1, v_2) \cdot \pi) \triangleq p \cdot \text{calls}(\pi)\)
  \(\text{calls}(p_1 \cdot p_2 \cdot \pi) \triangleq \text{calls}(p_2 \cdot p_1 \cdot \pi)\)
- On events: \(\text{calls}(\text{Ret}(\pi, a, v)) \triangleq \text{calls}(\pi, a, v)\)
  \(\text{calls}(\beta(\pi, v_1, v_2)) \triangleq \beta(\text{calls}(\pi, v_1, v_2))\)
- On traces: \(\text{calls}(\epsilon) \triangleq \epsilon\)
  \(\text{calls}(e, tr) \triangleq (\text{calls}(e), (\text{calls}(tr))\)

We deliberately drop the Call annotations, because our goal is to recover eventually the call strings that are used in \(k\)-CFA. We leave to future work the question of keeping Call annotations to obtain other forms of context sensitivity.

The second selection pass on events consists in keeping only events that are labelled by program points—as opposed to call annotations. This is simply done as follows:

**Definition 3.3** (Event filtering).

\[
\begin{align*}
\text{ret} \; \epsilon & \triangleq \epsilon \\
\text{ret}(\beta(\pi, v_1, v_2), tr) & \triangleq \beta(\pi, v_1, v_2), \text{ret} tr \\
\text{ret}((\text{Ret}(\pi, p, v)), tr) & \triangleq \text{Ret}(\pi, p, v), \text{ret} tr \\
\text{ret}((\text{Ret}(\pi, \text{Call}(v_1, v_2), v)), tr) & \triangleq \text{ret} tr
\end{align*}
\]

By composing the two filter functions, we obtain the abstraction function \(\alpha_e\) for events, together with its corresponding concretization \(\gamma_e\):

**Definition 3.4** (Event abstraction and concretization).

\[
\begin{align*}
\alpha_e(S) & \triangleq \{ (\text{ret} \circ \text{calls}(tr), v) \mid (tr, v) \in S \} \\
\gamma_e(S) & \triangleq \{ (tr, v) \mid \text{ret} \circ \text{calls}(tr) \subseteq tr' \land (tr', v) \in S \}
\end{align*}
\]

**Lemma 3.5.** The functions \(\alpha_e\) and \(\gamma_e\) form a Galois connection: \((\mathcal{P}(T \times V), \subseteq) \leq_{T, c} (\mathcal{P}(T \times V), \subseteq)\).

The proof relies on the monotonicity of calls and ret with respect to the sub-sequence order.

The function \(\gamma_t[tr]\) for a fully annotated term \(t\) satisfies the following inclusions:

**Lemma 3.6** (Semantic inclusions after event abstraction).

\[
\begin{align*}
\alpha_e([(x)]^p)_I & \subseteq \{ (\text{Ret}(e, p, \sigma(x)), \sigma(x)) \mid e \in I \} \\
\alpha_e([(\lambda x. t)]^p)_I & \subseteq \{ (\text{Ret}(e, p, (\lambda x. t) \cdot \sigma)), (\lambda x. t) \cdot \sigma)) \mid e \in I \} \\
\alpha_e([(\mathrm{t}_1])^p_1 [\mathrm{t}_2]^p_2)_I & \subseteq \begin{cases} 
(tr_1 + \text{tr}_2 + \\
\beta(p, \lambda x. [\mathrm{t}_1]^p_1, [\mathrm{t}_2]^p_2, v_2) \Rightarrow \\
\text{ret}(e, p, v_3) \Rightarrow \\
\text{ret}(\lambda x. [\mathrm{t}_1]^p_1, \{v_3 \in \alpha_e([\mathrm{t}_1]^p_1)_I \}
\end{cases}
\end{align*}
\]

Compared to Lemma 2.10, we notice that the events and contexts involved in a call annotation have been removed. In particular, the trace \(\text{tr}_1\) that corresponds to the evaluation of the function body obtained from \(\text{t}_1\) is now only prefixed by the call site \(p\), i.e. the program point of the application.

### 3.2 Unordered History Abstraction

In this abstraction step, we forget the order of events in the computation trace. Instead of a trace, we produce two maps that contain sets of values.
The first map is a global environment that inspects the \(\beta\)-reduction events and remembers on which arguments the functions were called, and in which context. This environment map is defined by the env function.

The second map is a global cache, that reads the return events and remembers which values were produced for the different program points, and in which context. This cache map is defined by the cache function.

**Definition 3.7 (Building environments and caches).**

\[
\begin{align*}
\text{env}(v) & \triangleq \emptyset \\
\text{env}(\text{Ret}(\delta, p, o), \text{tr}) & \triangleq \text{env tr} \\
\text{env}(\beta(\delta, x, t, o), \text{tr}) & \triangleq \{(v, t) \mapsto \{v\} \} + \text{env tr} \\
\text{cache}(v) & \triangleq \emptyset \\
\text{cache}(\text{Ret}(\delta, p, o), \text{tr}) & \triangleq \{(p, t) \mapsto \{v\} \} + \text{cache tr} \\
\text{cache}(\beta(\delta, v_1, v_2), \text{tr}) & \triangleq \text{cache tr}
\end{align*}
\]

The definitions of env and cache rely on the following operations on maps with a set-valued range.

**Definition 3.8 (Operations and relations on maps).**

\[
\begin{align*}
\emptyset & \triangleq \lambda k. \emptyset \\
\{k \mapsto S\} & \triangleq \lambda k'. \text{if } k' = k \text{ then } S \text{ else } \emptyset \\
\pi \cdot m & \triangleq \lambda (x, \pi'). \text{if } \pi' = \pi \cdot \pi'' \text{ then } m(\pi'') \text{ else } \emptyset \\
m_1 + m_2 & \triangleq \lambda k. m_1(k) \cup m_2(k) \\
m_1 \subseteq m_2 & \triangleq \forall k. m_1(k) \subseteq m_2(k)
\end{align*}
\]

The above definitions comprise the empty map, the singleton map, how to prefix a map with a path, how to take the union of two maps, and how to compare them. The definitions of history abstraction and concretization follow.

**Definition 3.9 (History abstraction and concretization).**

\[
\begin{align*}
\alpha_h(S) & \triangleq \{(\text{env tr}, \text{cache tr}, v) \mid (\text{tr}, v) \in S\} \\
\gamma_h(S) & \triangleq \{(\text{tr}, v) \mid \text{env tr} \subseteq \rho \land \text{cache tr} \subseteq C \land (\rho, C, v) \in S\} \\
\delta_1 \subseteq \delta_2 & \triangleq \forall (\rho, C_1, v_1) \in \delta_1 \exists (\rho_2, C_2, v_2) \in \delta_2 \mid \rho_1 \subseteq \rho_2 \land C_1 \subseteq C_2 \land v_1 = v_2
\end{align*}
\]

**Lemma 3.10.** \((\varphi(\Sigma \times \Sigma), \subseteq) \preceq_{\alpha_{\text{env}}} (\varphi(\Sigma(\Sigma) \times C(\Sigma) \times \Sigma), \subseteq)\)

The proof relies on the monotonicity of env and cache.

We write \(A_h = \alpha_0 \circ \alpha_e\) for the chaining of the abstractions we have defined so far. The abstract semantics of a fully annotated term \(A_h(\ term)\) satisfies the following inclusions:

**Lemma 3.11 (Semantic inclusions after history abstraction).**

\[
\begin{align*}
A_h([\ term])_{\text{tr}} & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\} \\
A_h([\ term]_{\text{tr}}) & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\} \\
A_h([\ term]_{\text{tr}}) & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\}
\end{align*}
\]

The call site \(p\) prefixes the environment \(\rho_3\) and cache \(C_3\) that result from the evaluation of the function body of the application. Moreover, the global environment records that the arguments of these functions contain the possible values for the arguments of the application. Finally, the global cache remembers that the whole term, i.e. the program point \(p\) in the empty environment, evaluates to the possible values of the function body.

### 3.3 Forgetting Output Values and Creation of Environments and Caches

The next abstraction steps are administrative, simple, and use mostly standard Galois connections. We only give a brief sketch of how they work.

First, we only keep the environment and the cache, using the abstraction function \(A_{\text{NoVal}}(S) \triangleq \{ (\rho, C, v) \mid \exists v. (\rho, C, v) \in S \}\). Then, we separate environments from caches: using the function \(A_\Lambda(S) \triangleq \{ (\rho \mid \exists C, (\rho, C) \in S), (C \mid \exists \rho, (C, \rho) \in S)\}\), we transform sets of pairs of maps into sets of pairs of maps. Finally each set—of environments or caches—is abstracted as a map—an environment or a cache, respectively—using the function \(A_{\Sigma} \triangleq \lambda k. \bigcup_{m \in M} m(k)\) to abstract the maps, and \(A_{\Sigma} C(S_1, S_2) \triangleq \{\text{eval}(S_1), \text{eval}(S_2)\}\) to abstract their pairs. All these functions are adjoints of Galois connections.

We define the function \(A_{EC} \triangleq A_{\Sigma} \circ A_\Lambda \circ A_{\text{NoVal}} \circ A_h\), that stacks up all the abstractions we have defined so far. The semantic inclusions for the abstract semantics of a fully annotated term \(A_{EC}([\term])_{\text{tr}}\) are easily obtained, by noticing that for a term \(\term\) with an output value \(v\) and a cache \(C\) obtained by \(A_h([\term])_{\text{tr}}\), we always have \(\{v\} \subseteq C(p, e)\).

### 3.4 Abstracting Inputs

The next abstraction abstracts the set of inputs \(I\). We transform this set of substitutions—i.e. maps from variables to values—into a map from variables to sets of values, using the Galois connection \(\preceq_{\gamma_{\text{tr}}}\) we defined in the previous section. We denote by \(E\) the maps from variables to sets of values. The abstract semantics \(A_{EC}([\term])_{\text{tr}}\) of a fully annotated term \(\term\) satisfies the following inclusions:

**Lemma 3.12 (Semantic inclusions after input abstraction).**

\[
\begin{align*}
A_{EC}([\term]_{\text{tr}}) & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\} \\
A_{EC}([\term]_{\text{tr}}) & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\} \\
A_{EC}([\term]_{\text{tr}}) & \subseteq \{(\text{tr}, \{v \mapsto \{\text{eval}(x)\} \mid x \in I\} \mid \sigma, \sigma \in I\}
\end{align*}
\]

The call site \(p\) prefixes the environment \(\rho_3\) and cache \(C_3\) that result from the evaluation of the function body of the application. Moreover, the global environment records that the arguments of these functions contain the possible values for the arguments of the application. Finally, the global cache remembers that the whole term, i.e. the program point \(p\) in the empty environment, evaluates to the possible values of the function body.
environment. In other words, it is a closure with a set of possible closing environments. We abstract these sets of closures in the next section.

For applications, the environments and caches are built by combining the environments and caches that are computed for each sub-term of the application with the ones that are obtained for the functions \( \lambda x. [t_1]p \) that may be called. The environment \( \rho_3 \) records which arguments were given to the possibly called functions, whereas the cache \( \mathcal{C} \) reports that the result of the whole application contains the results of the execution of every executed function body. Interestingly, the terms \([t_1]p \)
 are analysed in a context that does not involve the initial context \( \gamma_m(E) \), because \( \lambda x. [t_1]p \) is a closed value, and therefore the body \([t_1]p \)
 needs no other input than one for its argument \( x \). For this argument \( x \), we provide the possible values for the arguments of the call.

### 3.5 Closure Abstraction

In this abstraction step, we introduce an abstract domain to represent sets of closures. An abstract closure is composed of a value—a \( \lambda \)-abstraction—and an environment that assigns to each of the free variables of the value a set of abstract closures. The definition of this abstract domain is recursive. This abstract domain will be the starting point of the new analysis \( \mathcal{V} \mathcal{C} \mathcal{F} \) we can describe in §5.

#### Definition 3.13 (Abstract closures and their concretization)

\[
\text{Clos} \triangleq \forall \times \mathcal{G} \quad \mathcal{G} \triangleq \mathcal{X} \xrightarrow{\text{fn}} \Phi(\text{Clos})
\]

\[
\gamma_0 \triangleq \forall \gamma \circ \gamma_0 \quad \gamma_{\text{clos}} \triangleq \text{map}_{\gamma_{\text{clos}}}
\]

\[
\gamma'_{\text{clos}}(S) \triangleq \bigcup_{(x,t) \in S} \{(x,t) \sigma \mid \sigma \in \gamma_0(\Gamma)\}
\]

The concretization of a set of closures builds the set of values for each closure in the set by closing over all the possible environments, and then takes the union of those sets. The inclusions for the abstracted semantics follow:

#### Lemma 3.14 (Semantic inclusions after closure abstraction).

\[
\begin{align*}
\text{AEC}(\forall[x]p)_{\gamma(\Gamma)} & \preceq \gamma_{\text{clos}}(\emptyset, \{(p, e) \mapsto \Gamma(x)\}) \\
\text{AEC}(\forall[x,t]p)_{\gamma(\Gamma)} & \preceq \gamma_{\text{clos}}(\emptyset, \{(p, e) \mapsto \{\langle x,t \rangle \mid \Gamma([x,t], x, t)\}\})
\end{align*}
\]

#### Definition 3.15 (Indirection abstraction).

\[
\text{Clos} \subseteq \forall \times \mathcal{G} \quad \gamma_\text{clos} \triangleq \forall \gamma \circ \gamma_0 \quad \gamma'_{\text{clos}}(S) \triangleq \{\{c \mid \{c \in \gamma_{\text{clos}}(S) \} \cap \gamma_0(\Gamma)\}
\]

The meaning of an abstract closure depends on a global environment \( \hat{\rho} \in \mathcal{R}(\text{Clos}) \), that maps pairs \((t, \delta)\) to sets of abstract closures with indirection. An abstract closure with indirection denotes the set of the abstract closures that are obtained by unfolding the definitions contained in \( \hat{\rho} \) a finite number of times. We formally define this in Def. 3.15, using the inductively defined relation \( \epsilon^\text{ind} \) that performs these unfoldings. Our concretization function \( \gamma_{\text{clos}}^\hat{\rho} \) transforms a set of abstract closures in \( \text{Clos} \) into a set of abstract closures in \( \text{Clos} \) from §3.5. Our definitions are inspired from the correctness relation \( R \) of [33, p. 159]—an inductive relation that describes the concrete closures that are obtained by finitely unfolding indirections.

The semantic inclusions that result from the indirection abstraction are similar to those of Lemma 3.14.

#### Lemma 3.16 (Inclusions after indirection abstraction).

\[
\begin{align*}
\text{AEC}(\forall[x]p)_{\gamma(\hat{\rho}, \hat{\Gamma})} & \preceq \gamma_{\text{clos}}^\hat{\rho}(\emptyset, \{(p, e) \mapsto \hat{\rho}(\Gamma(x))\}) \\
\text{AEC}(\forall[x,t]p)_{\gamma(\hat{\rho}, \hat{\Gamma})} & \preceq \gamma_{\text{clos}}^\hat{\rho}(\emptyset, \{(p, e) \mapsto \{\langle x,t \rangle \mid \Gamma([x,t], x, t)\}\})
\end{align*}
\]

3.6 Indirection Abstraction

The indirection abstraction is the most complex of this paper, and constitutes a crucial step to eventually recover the textbook definition of \( k\)-CFA. The idea of the abstraction is to introduce an indirection through names to break the recursion cycle in the definition of closures. Here, we choose call strings as names. Previous work from the AAM family have developed this idea too, also to break circularity. They baked the indirection in their semantics using a global store.

The new abstract closures \( \check{\epsilon} \in \text{Clos} \) are composed of a \( \lambda \)-abstraction and an environment \( \hat{\Gamma} \in \hat{\mathcal{G}} \) that assigns to each free variable a binding label \( \ell \) and a name \( \delta \) as opposed to a set of closures as done in §3.5. The binding label refers to the binding site of the variable, and the name \( \delta \) refers to the context in which the variable has been assigned to a value.

#### Definition 3.15 (Indirection abstraction).

\[
\text{Clos} \subseteq \forall \times \mathcal{G} \quad \gamma_\text{clos} \triangleq \forall \gamma \circ \gamma_0 \quad \gamma'_{\text{clos}}(S) \triangleq \{\{c \mid \{c \in \gamma_{\text{clos}}(S) \} \cap \gamma_0(\Gamma)\}
\]

The meaning of an abstract closure depends on a global environment \( \hat{\rho} \in \mathcal{R}(\text{Clos}) \), that maps pairs \((t, \delta)\) to sets of abstract closures with indirection. An abstract closure with indirection denotes the set of the abstract closures that are obtained by unfolding the definitions contained in \( \hat{\rho} \) a finite number of times. We formally define this in Def. 3.15, using the inductively defined relation \( \epsilon^\text{ind} \) that performs these unfoldings. Our concretization function \( \gamma_{\text{clos}}^\hat{\rho} \) transforms a set of abstract closures in \( \text{Clos} \) into a set of abstract closures in \( \text{Clos} \) from §3.5. Our definitions are inspired from the correctness relation \( R \) of [33, p. 159]—an inductive relation that describes the concrete closures that are obtained by finitely unfolding indirections.

The semantic inclusions that result from the indirection abstraction are similar to those of Lemma 3.14.
The semantic inclusions exhibit one degree of freedom, in the case of applications. The name \( \delta(\hat{c}) \) to assign to the arguments of function bodies can be chosen freely, as long as the global environment satisfies that the possible values for the argument in the context \( \delta(\hat{c}) \) contains the possible values of \( t_2 \). This degree of freedom has been thoroughly explored by the AAM line of work, through the use of specific allocation functions. In CFA, the standard choice is to take the current enclosing context extended with \( p \) (the call site).

In §4, we will exploit the inclusions of Lemma 3.16 to justify the constraint rules for \( \infty \)-CFA.

3.7 Call Sites Abstraction: Context Projection

The final abstraction step is to lose information by identifying certain contexts. This is where standard CFAs transform the search space into a finite one. We model this collapse of the search space by a projection function \( \phi \in \mathcal{D} \to \mathcal{D} \).

**Definition 3.17** (Context projection abstraction).

\[
\begin{align*}
\alpha_\mathcal{D}(\phi)(m) & \triangleq \lambda(x, \delta). \bigcup_{(\phi(\delta'), \delta) = \delta'} m(x, \delta') \\
\gamma_\mathcal{D}(\phi)(m) & \triangleq \lambda(x, \delta). m(x, \phi(\delta))
\end{align*}
\]

The action of \( \phi \) on maps is to unite the sets that are assigned to the keys \( k_1 \) and \( k_2 \), when \( \phi(k_1) = \phi(k_2) \). So far, no hypothesis on \( \phi \) is required. The functions \( \phi \) define the context sensitivity policies of the analysis. By taking \( \phi(\delta) = \delta \), we keep the same abstract semantics, i.e. \( \infty \)-CFA. If we take \( \phi(\delta) = \varepsilon \), there is no distinction between contexts anymore, and we obtain \( 0 \)-CFA. To get \( k \)-CFA, we take \( \phi(\delta) = [\delta]_k \), i.e. the function that keeps the last \( k \) elements of \( \delta \).

**Lemma 3.18.** \( \alpha_\mathcal{D}(\phi) \) and \( \gamma_\mathcal{D}(\phi) \) form a Galois connection: \( \left( \mathbb{R}(\text{Clos}) \times \mathbb{C}(\text{Clos}), \alpha_\mathcal{D}(\phi) \right) \leq \alpha_\mathcal{D}(\phi) \left( \mathbb{R}(\text{Clos}) \times \mathbb{C}(\text{Clos}), \gamma_\mathcal{D}(\phi) \right) \)

4 Constraint-Based CFA

As a first illustration of the use of the abstractions defined in the previous sections we show how they enable a semantic and modular soundness proof of the constraint rules for infinitary CFA—where no approximation of contexts is performed—and for \( \hat{\phi} \)-CFA, a family of analyses parameterised by a projection function \( \phi \), that generalises uniform \( k \)-CFA. More specifically, we give a semantic definition of what it means for a pair \((\hat{\rho}, \hat{C})\) to be a solution of a CFA problem. As a consequence, the syntactic constraint rules become theorems. A benefit of this approach is that the proof method is modular: each rule is proved sound separately, instead of considering the set of rules as a whole.

We use the same notation \((\hat{\rho}, \hat{C}) \models^\mathcal{D} [t]^p\) as in [33, p. 190], that defines the valid solutions of a CFA problem. This judgement denotes that \((\hat{\rho}, \hat{C})\) is a valid solution for the CFA of the term \([t]^p\), taken in the calling context \( \delta \) and in the environment \( \hat{\Gamma} \). Our semantic definition follows:

**Definition 4.1** (Semantic validity of an \( \infty \)-CFA solution).

\( (\hat{\rho}, \hat{C}) \models^\mathcal{D} [t]^p \triangleq \delta \cdot \mathcal{A}_{\text{TEC}}([t]^p) \models^2 \gamma_{\text{clos}} \circ \gamma_{\text{ind}}(\hat{\rho}, \hat{C}) \)

The validity of the solution \((\hat{\rho}, \hat{C})\) is stated as an over-approximation property. Consider the set of inputs \( I \) defined by \( \hat{\Gamma} \) under the definitions of \( \hat{\rho} \), i.e. \( I = \gamma_{\text{clos}}(\hat{\rho}, \hat{\Gamma}) \). The definition tells us that \((\hat{\rho}, \hat{C})\) must be an over-approximation of the semantics of the program \([t]^p\) for the set of inputs \( I \), where this semantics is considered “in the calling context \( \delta \).” This is why the maps obtained by abstracting the semantics are prefixed with the call site path \( \delta \).

From Lemma 2.9 and the monotonicity of the abstractions it follows that semantic validity is preserved by reduction:

**Lemma 4.2.** If \( t_1 \twoheadrightarrow t_2 \) and \((\hat{\rho}, \hat{C}) \models^\mathcal{D} t_1 \), then \((\hat{\rho}, \hat{C}) \models^\mathcal{D} t_2 \).

The proof method we employ for the preservation property radically differs from previous proofs used in constraint-based approaches [16, 17, 33, 46], where preservation is proved in a syntactic manner, by reasoning globally on the whole set of rules that define the constraints: their proofs inspect complete derivation trees built from the rules for the initial program, and reconstruct derivation trees for the reduced program. Such proofs follow the style of subject reduction proofs for type systems [35], and are hard to maintain and to extend. In contrast, our proof of preservation is short and simple, and is independent of the set of rules, as it relies solely on the definition of semantic validity and on the monotonicity of the abstractions.

Our central result is the following theorem, that asserts the soundness of the rules for \( \infty \)-CFA.

**Theorem 4.3** (Constraints for \( \infty \)-CFA). The following inference rules are sound:

\[
\begin{align*}
\phi(\Gamma(x)) & \subseteq \hat{C}(p, \delta) & \{\langle x.t, \hat{\Gamma}\rangle_{\text{flat}(x.t)}\} & \subseteq \hat{C}(p, \delta) \\
\phi(\Gamma(x)) & \models^\mathcal{D} [x]^p \quad \hat{\rho}(\hat{\Gamma}(x)) & \models^\mathcal{D} [x]^p
\end{align*}
\]

\[
\begin{align*}
(\hat{\rho}, \hat{C}) & \models^\mathcal{D} [t_1]^p & (\hat{\rho}, \hat{C}) & \models^\mathcal{D} [t_2]^p \\
\forall \langle x.t, [t_1]^p, \hat{\Gamma}\rangle & \in \hat{C}(p_1, \delta), \\
(\hat{\rho}, \hat{C}) & \models^\mathcal{D} [x.t]^p & \wedge \hat{C}(p_2, \delta) & \subseteq \hat{\rho}(p, \delta) \wedge \hat{C}(p_3, \delta p) & \subseteq \hat{C}(p, \delta)
\end{align*}
\]

\[
(\hat{\rho}, \hat{C}) \models^\mathcal{D} \{\langle [t_1]^p \rangle_{\text{flat}(t_2)}\}^p
\]

The proof relies on the inclusions provided by Lemma 3.16. The main arguments to conduct the proof are the monotonicity of the path-prefixing operator, and the fact that both path-prefixing and map-union semi-commute with the concatenations \( \gamma_{\text{clos}} \) and \( \gamma_{\text{ind}} \) (i.e. \( \delta \cdot \gamma(m) \leq \gamma(\delta \cdot m) \) and \( \gamma(m_1) + \gamma(m_2) \leq \gamma(m_1 + m_2) \)) for the order relations of interest). Importantly, each rule is proved sound independently of each other. We believe this makes sound control-flow analyses easier to design, and also more robust to extend with new features. To our knowledge, this is the first compositional explanation of a context-sensitive CFA.

By proceeding in a similar way as for \( \infty \)-CFA, we can generalise the previous results to the validity of \( \hat{\phi} \)-CFA, i.e. a CFA problem where the calling contexts are projected by
a function $\phi$. The statement differs from Def. 4.1 only by adding a call to the abstraction function $\alpha_{\omega}(\phi)$ from §3.6.

**Definition 4.4** (Semantic validity of a $\phi$-CFA solution).

$$(\phi, \hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [t]P \iff 
\alpha_{\omega}(\phi)(\delta \cdot \mathcal{E}C([t]P; \gamma_{\delta} (\hat{\rho}, \hat{\mathcal{C}})) \subseteq \hat{\mathcal{C}}_{\text{clo}} \circ \gamma_{\text{fin}} (\hat{\rho}, \hat{\mathcal{C}})$$

The following theorem follows. If we take $\phi(\delta) = [\delta]_k$, the theorem gives a geometric justification for the textbook constraint rules of uniform $k$-CFA.

**Theorem 4.5** (Constraints for $\phi$-CFA). Assume that $\phi \circ \phi = \phi$ and that $\delta$ and every path that occurs in $\hat{\rho}, \hat{\mathcal{C}}, \hat{\Gamma}$ are fixpoints of $\phi$. Then, the following inference rules are sound:

$$\frac{\hat{\rho}(\Gamma(x)) \subseteq \hat{\mathcal{C}}(p, \delta) \quad \{\hat{\chi} x. t, \Gamma(\hat{\chi} x. t, 1)\} \subseteq \hat{\mathcal{C}}(p, \delta)}{(\hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [\hat{\chi} x. t]P}$$

$$\frac{(\hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [t_1]P_1 \quad (\hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [t_2]P_2 \quad \forall(\hat{\chi} x. [t_1], P_1, \Gamma_1) \in \hat{\mathcal{C}}(P_1, \delta), \quad (\hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [\hat{\chi} x. \phi(\delta p)]P_1 \quad \land \hat{\mathcal{C}}(p, \delta) \subseteq \hat{\mathcal{C}}(p, \delta) \land \hat{\mathcal{C}}(p, \delta) \subseteq \hat{\mathcal{C}}(p, \delta)}{(\hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [[t_1]P_1][t_2]P_2}$$

As expected, the rules differ from those of $\omega$-CFA only in the application rule, where the projection $\phi$ is used to project the extended calling context $\delta p$ during the analysis of the function bodies. The idempotency hypothesis $\phi \circ \phi = \phi$ ensures that the new paths $\phi(\delta p)$ are fixpoints of $\phi$, which is made necessary by our requirement that $\hat{\rho}$ and $\hat{\mathcal{C}}$ should only contain paths that are fixpoints of $\phi$.

In $k$-CFA, the chosen $\phi$ is such that $\phi$ has a finite range. This ensures that there is a finite number of $\Gamma$, $\hat{\rho}$ and $\hat{\mathcal{C}}$. Consequently, the resolution of the system of constraints necessarily explores a finite number of states, because the set of states has been made finite. In §5, we consider instead a system where the set of states is infinite, but where the algorithm is guaranteed to explore a finite number of them. This is achieved by widening techniques.

To conclude this section, we give the precise semantic property that a solution to $\phi$-CFA enjoys.

**Theorem 4.6**. Assume $(\phi, \hat{\rho}, \hat{\mathcal{C}}) \models^{\hat{\phi}} [t_0]P_0$ and $(t_0)_{P_0} \xrightarrow{\text{tr}} v_0$ and $t_0$ is a closed program. Then:

- If $\beta(\pi, x, t, v) \in \text{tr}$, then there exists an abstract closure $\hat{c} \in \hat{\rho}(t, (\phi(\text{calls}(\pi))))$ such that $v \in \gamma_{\text{fin}}(\hat{c})$.
- If Ret$(\pi, p, v) \in \text{tr}$, then there exists an abstract closure $\hat{c} \in \hat{\mathcal{C}}(p, (\phi(\text{calls}(\pi))))$ such that $v \in \gamma_{\text{fin}}(\hat{c})$.

The theorem highlights the meaning of $\hat{\rho}$ and $\hat{\mathcal{C}}$ as solutions of the $\phi$-CFA analysis of the closed program $t_0$ considered in the empty context. First, the theorem tells us that for every $\beta$-reduction event in the trace of the analysed program, there is a corresponding entry in the global environment $\hat{\rho}$ that contains an abstract value $\hat{c}$ that contains the value used as argument during that $\beta$-reduction. Second, for every return event in the trace of the program, there is in the cache $\hat{\mathcal{C}}$ a corresponding entry that contains an abstract value $\hat{c}$ that contains the returned value as reported by the event.

The proof of Theorem 4.6 proceeds by unfolding Definition 4.4 and exploiting the order relations and the concretization functions of the Galois connections.

## 5 Beyond the Finite State Abstraction

In this section, we deviate from the finite state approximation methodology usually applied in CFA, and introduce an abstract domain for closures, that is recursively defined, and whose values can have unbounded depths. To ensure the convergence of the analysis, we use a widening operator, as prescribed by the abstract interpretation theory, that makes ascending chains stationary. We obtain the $\text{VCF}$ analysis.

### 5.1 Abstract Closures with Widening

The abstract domain is derived from the domain of Def. 3.13. The environments in that previous domain could contain infinite sets of closures. In this new domain, we only consider finite sets of closures, and moreover, we enforce that there is at most one abstract environment per function. The abstract closures $c^d$ in $\text{Clos}^d$ are thus either a finite map from functions to their closing abstract environment, or the element $\top$, that represents the set of all closures from §3.5.

**Definition 5.1** (Abstract Closures).

$$\text{Clos}^d \triangleq (\forall \text{fin} \xrightarrow{\text{dom}} \{\top\}) \quad \{\top\} \triangleq \bigcup_{\text{fin}} \text{Clos}^d \quad \text{Clos}^d_{\text{clo}}(m) \triangleq \bigcup_{\text{fin}} \{\text{map} \in \text{Clos}^d_{\text{clo}}(\Gamma^d)\}$$

We define the union of abstract closures as follows:

**Definition 5.2** (Union of closures).

$$\text{Clos}^d_{\text{clo}} \triangleq \bigcup_{\text{fin}} \text{Clos}^d_{\text{clo}} \quad \text{Clos}^d_{\text{clo}} \triangleq \bigcup_{\text{fin}} \text{Clos}^d_{\text{clo}} \quad m \triangleq \bigcup_{\text{fin}} \text{Clos}^d_{\text{clo}}$$

$$m_1 \cup_{\text{fin}} m_2 \triangleq m_1 \cup_{\text{fin}} m_2$$

The interesting part of the definition is for the finite maps, that represent the set of closures. The union is defined as the pointwise union of these maps.

Using solely this abstract union operator in an abstract interpreter could make the analysis diverge, because unions can build infinite ascending chains of values. Therefore, we introduce a widening operator $\text{widen}^d_{\text{clo}}$, that is more approximate than $\text{widen}^d_{\text{clo}}$, and ensures that sequences of abstract values built from widenings are eventually stationary. The widening operator is defined as follows:
The abstract closure stack is depicted in the following diagram: \[ T \rightarrow (X \rightarrow \phi(Clos^\#)) \rightarrow Clos \]

- \( \gamma_{\text{Cache}}(p) \rightarrow \alpha_{\text{Cache}}(p) \)
- \( \gamma_{\text{NoEnv}} \rightarrow \alpha_{\text{NoEnv}} \)
- Cache projection
- Global environment removal

Here, \( Clos \) represents the cache as an intermediate result.

The abstractions are defined recursively:

\[ m\_1 \vdash m\_2 \rightarrow \lambda v. m\_1(v) \quad \text{if } v \in \text{domain } m\_1 \cap \text{domain } m\_2 \]
\[ m\_2 \vdash m\_1(v) \quad \text{if } v \in \text{domain } m\_1 \setminus \text{domain } m\_2 \]
\[ I\_\Gamma{x}\_\Delta{x}^\# \vdash \lambda x. I\_\Gamma{x}\_\Delta{x}^\#(x) \]

The input widening function \( h(m) \), the height of the finite map, is the maximum of the heights of the elements in the range of \( m \). The definition also involves the function \( I^\#_n \), that truncates each element in the range of \( I^\# \) to the height \( n \), i.e., replaces sub-structures with \( T \), once the depth \( n \) is reached in the structure. The interesting case of the definition is for maps: to compute \( m\_1 \vdash m\_2 \), the bindings in \( m\_2 \) that are not already present in \( m\_1 \) are truncated to the height of \( m\_1 \). Although the map \( m\_1 \vdash m\_2 \) can be wider than \( m\_1 \), its height cannot be larger than \( m\_1 \)'s. Because the possible keys of these maps are in finite number (the keys are the \( As \) occurring in the program), the widening chains are eventually stationary.

### 5.2 Abstractions Towards \( \nabla \text{CFA} \)

The different abstraction steps that lead to \( \nabla \text{CFA} \) are summarized in Fig. 3. The starting point is the abstract semantics we obtained after closure abstraction (§3.5). From there, we apply our new abstraction, that builds the new abstract values of §5.1. Then, we notice in the inclusions of Lemma 3.14 that the global environment \( \rho \) is not necessary to compute the cache. So, we remove the global environment \( \rho \), using a standard abstraction. Finally, we choose to only keep the value of the cache that corresponds to the whole program that is analysed, i.e., for the program \( [t]P \), we only keep the entry \( C(P) \). In §5.3 we present a solver that computes such a cache as an intermediate result.

We define \( G(p) \equiv \gamma_{\text{Cache}} \circ \gamma_{\text{Clos}} \circ \gamma_{\text{NoEnv}} \circ \gamma_{\text{Cache}}(p) \) and \( G_\Delta \equiv \gamma_{\Delta} \circ \text{map}_{\phi} \). At the top of the abstraction stack, the following semantic inclusions hold:

#### Lemma 5.4 (Semantic inclusions for \( \nabla \text{CFA} \))

\[ A_{E\Delta}(\alpha(x, t)|_{\Gamma^\#}) \equiv G(p)(\Gamma^\#(x)) \]
\[ A_{E\Delta}(\alpha(x, t)|_{\Gamma^\#}) \equiv G(p)(\alpha(x, t)|_{\Gamma^\#}) \]

where \( A_{E\Delta}(\alpha(x, t)|_{\Gamma^\#}) \equiv G(p)(\alpha(x, t)|_{\Gamma^\#}) \)

and \( \nabla \) is an \( \nabla \text{CFA} \) abstraction steps to obtain \( \nabla \text{CFA} \).

The inclusions resemble a big-step evaluator that computes sets of values. For variables, a lookup is performed in the environment. For functions, a singleton containing an abstract closure is returned. When functions are found at an application point, each of them is analysed, and the union of their analysis results is returned.

### 5.3 A Widening-Based Control Flow Analysis

In this section, we give some elements about the implementation of \( \nabla \text{CFA} \), that is publicly available as a companion artefact. The core of the analyser is concentrated in the following function, named analyze:

#### Definition 5.5 (\( \nabla \)-analysis)

\[ \text{analyze}(\text{analyze}_{\text{rec}})(iw, \delta, I^\#, [x]P) \equiv T^\#(x) \]
\[ \text{analyze}(\text{analyze}_{\text{rec}})(iw, \delta, I^\#, [x]P) \equiv \langle I^\#, x, \Delta^\# \rangle \]

analyze(\text{analyze}_{\text{rec}})(iw, \delta, I^\#, [x]P) in

\[ \text{let } V_1 = \text{analyze}_{\text{rec}}(\text{false}, \delta, I^\#, [t]P) \text{ in} \]

\[ \text{let } V_2 = \text{analyze}_{\text{rec}}(\text{false}, \delta, I^\#, [t]P) \text{ in} \]

\[ \text{let } iw' = \text{maximal}(\phi(\Delta)) \text{ in} \]

\[ \bigcup_{\langle x, t, I^\#, \Delta \rangle \in V_1} \text{analyze}_{\text{rec}}(iw', \phi(\Delta), (I^\#, x, V_2), [t]P) \]

analyze : (bool \times \Delta \times \Delta^\# \times T) \rightarrow Clos^\# \equiv \mu(\text{analyze})

The program analyze is executed using a top-down solver, i.e., a higher-order function \( \mu(f) \) (\( (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \)) that computes a function \( \mu(f) \) such that \( f(\mu(f))(x) \leq \mu(f)(x) \). The functional \( f \) is written using the open recursion style, that was also highlighted in [4]. The actual computation \( \mu(f)(x) \) on some argument \( x \) is performed on demand. A simple top-down solver memoizes the calls to \( f \) together with the so-far computed outputs, and iterates until it finds a post-fixpoint: it computes a cache of the calls, that is similar to the cache structure used in CFAs. For this paper, we implemented a simple, non-optimised top-down solver, that we extended so that input widening can be demanded by the client function \( f \) using the boolean argument \( iw \), as we explain later in the section. More efficient top-down solvers, where computation is incremental, have already been described in the literature [1, 5, 38].

Our analyser is parameterised by a projection function \( \phi \), that is required to have a finite orbit:
We name this strategy with a recursive call to analyze widening sequences, which implies the termination of the analyzer. When the solver detects a cycle, the domain's union is used for widening in the remainder of the document. As a counter-example, the identity function does not satisfy the finite orbit property.

The function named maximal in our code tests whether an element belongs to the set of maximal elements of .

The argument iw of the analyzer controls whether input widening must be performed by the solver. When confronted with a recursive call, analyze_rec (true, δ, t) — i.e. when input widening is demanded by the client—the top-down solver searches for previous calls with the arguments δ and t, and computes the widening of those t*, together with Γ*. This new, widened environment, is then used as argument to actually perform the recursive call.

The termination of the algorithm is ensured by the combination of two facts. First, the top-down solver performs widening for the outputs of the function. To keep the results accurate enough, widening on outputs is performed only when the solver detects a cycle; the domain’s union is used otherwise. Second, the inputs are widened when iw = true. Thus, it is the responsibility of the client function—here, the function analyze—to ensure that enough widenings on inputs are demanded to ensure convergence. Therefore, the termination of the analysis is obtained thanks to the cooperation between the top-down solver and its client function.

The first two recursive calls on the sub-terms [t₁]p and [t₂]p make the term argument decrease, and are thus not a source of divergence. The only source of non-termination are the recursive calls on the body of the computed closures. There are two cases for such calls: if δ(δ*p) is not maximal, then the number of annotations p' that can be added before the sequence of call sites reaches the maximal set decreases. This ensures convergence in this case. Otherwise, δ(δ*p) is maximal, and remains in the set of maximal elements, that is finite. Widening on Γ* is performed, and the stationarity of widening sequences implies the termination of the analyzer.

The call site paths that we use in the analyzer are only used as elements of a heuristic to guide the widening strategy. They are not part of the abstract values, as it is the case in k-CFA. Thus, these paths need no specific semantic justification for the soundness of the analysis to be proved.

We implemented k-CFA and ∇CFA in OCaml [23], so as to compare their precision and cost. For the comparison between the different analyses to be fair, we removed the computation of the cache from k-CFA (so that only the global environment and the abstract value for the whole program are computed), following what we did in §5. We extended the two analyses to support booleans and integers, so that more examples can be analyzed. The prototype analysers and the suite of program tests are publicly available in the artefact that accompanies the article.

With these experiments, we want to show that the area of the CFA design space we have uncovered is worth exploring more thoroughly: the experimental results that we obtain on standard examples extracted from the literature [8, 14, 18, 22, 27, 42] are encouraging us in pursuing in this research direction. Our goal is not to show that ∇CFA is better than existing CFAs, and more realistic benchmarks, e.g. measures on the impact of optimizations for compilers for functional languages, would be required to do so.

We provide in Table 1 an experimental comparison of the standard 0-CFA, the -CFA with the two strategies of limiting paths to the last call site (1-CFA) and the one limiting to at most one occurrence of call sites (1-CFA). We also compare with our widening-based analysis ∇CFA, using the same two strategies (1-∇CFA and 1-∇CFA). We are thus able to observe both the impact of the choice of φ and of the choice of standard CFA versus ∇CFA.

We perform the following measurements: we count how many states are needed for the top-down solver to explore the abstract call graph (column “S”), and how many edges this call graph contains (column “E”). These two measures are independent of the quality of our implementation: this graph remains the same, however incremental or efficient the solver is. The third measure is the number of iterations that are needed to find a post-fixpoint (column “Iter”). Because our solver does not try to avoid unnecessary recomputations, this measure can be seen as an upper bound of the iterations that an optimized solver would perform. Lastly, the fourth measure (column “Result”) is the obtained abstract value, that we compare for precision, taking 1-CFA as a baseline.

The main takeaways from the experiments are twofold:

1. -CFA generally yields better results than 1-CFA, because information loss on context sensitivity only happens when recursion is involved. For all but one program (church), -CFA improves over 1-CFA in terms of precision. Moreover, the sizes of the call graphs for 1-CFA are generally on par with those of 1-CFA, and so are the number of necessary iterations.

2. 1-∇CFA generally yields better results than 1-CFA and 1-CFA, in particular on programs that involve arithmetic computations, because classic CFAs cannot exploit expressive numeric domains such as intervals.
The use of widening generally makes the analysis converge faster but returns a less precise result. The fact that we lose precision with VCFA only in a few cases is indication that widening can be used with benefit here.

The hmca* examples from [14]—hmca stands for Hintze McAllester—show the expected blow-up of 0-CFA. As expected, 1-CFA performs better than 0-CFA on these examples, and gives the most accurate result. Remarkably, the other analyses explore the exact same graphs and therefore give the same results. In particular, the 1* strategy incurs no performance penalty and VCFA does not degrade precision.

One advantage of VCFA is that it removes the finiteness constraint on the height of the abstract domain, and thus allows the use of more expressive domains, such as intervals. This benefit is visible in several examples: with VCFA, the factorial fact is shown to return a result that is greater than 1, and the 91 function of McCarthy mc91 is shown to produce a result that is above 91. This is the best precision that can be achieved using intervals. This level of precision is not reachable by finite-height domains.

The examples sat, sat3 and sat4 are boolean satisfiability tests, that are applied to a boolean formula encoded as a function. Both 1-CFA and 1-VCFA fail to achieve the best result because the depths of the call stacks is statically bounded. In contrast, 1*-CFA and 1*-VCFA compute the best result—i.e. the formula is satisfiable—because the call stack is only bounded for recursively called program points.

The programs ack_cps, mc91_cps, tak_cps recursively produce a continuation—a higher-order value. Classic CFA is able to show that the three examples produce integers. VCFA analyses, however, fail to keep information on the results of the continuations, and are not able to prove that the outputs must be integers. This is due to the fact that our abstract domain for closures cannot represent recursively defined sets of closures, whereas the indirections used in k-CFA can express such solutions. Exploring ways to recover more precision is left for future work. We could for instance perform a narrowing pass, or design a more expressive domain for closures, that keeps information about the results of functions—a sort of function summary—or use a relational domain such as the one of [30], that keeps relations between inputs and outputs of functions.

7 Related Work

Control flow analysis for the lambda calculus was developed by Jones [19]. Shivers [41, 42] proposed a control flow analysis for Scheme programs in continuation-passing style and introduced the notion of k-CFA in which flow information is given relative to a context which represents an abstraction of the state. Sestoft [39] developed an analysis for computing the flow of closures in higher-order functional programs. The constraint-based formulation of CFA was introduced by Falsberg [34] and developed further by Nielsen and Nielson [31–33]. In their infinitary control flow analysis, they propose a constraint-based framework from which they show how to obtain different kinds of k-CFA by varying the notion of context. In that article, they leave it as an open
problem to relate the infinitary CFA to a collecting semantics of an operational semantics using abstract interpretation.

Shivers [42] outlines the proof of correctness of a control flow analysis for a CPS-based intermediate representation of Scheme. The proof is based on a non-standard denotational semantics which, in addition to the program’s result, computes a cache for each call site. This cache contains the identifiers of the lambda expressions called at that call site. From this semantics, it is possible to obtain 0-CFA and 1-CFA as abstract interpretations.

Jones and Rosendahl [21] give a denotational semantics for a higher-order functional language based on minimal function graphs [20]. This semantics computes a subset of the graph of a function (i.e., a set of (argument,result) pairs) that are necessary and sufficient to evaluate a given function call. This semantics is then abstracted in order to obtain a closure analysis that approximates the control flow of the program. The minimal function graphs could be obtained in our approach, by exploiting the return events that are labelled with call annotations—these events were removed by the first abstraction steps of §3.

Midtgaard and Jensen show how to use the principles of abstract interpretation to systematically derive control flow analyses from collecting semantics induced by abstract machines. They obtained context-insensitive analyses—a k to 0-CFA—for continuation-passing style [26] and for direct-style programs in ANF [27] based on the CESK machine.

Van Horn and Might [15] apply abstract interpretation to the internal state representation of a series of abstract state machines. This results in “abstract abstract machines” that compute an over-approximation of control flow and other intentional information about an execution. In machines with store-allocated continuations, control flow analysis becomes the problem of abstracting the store.

Darais et al. [4] demonstrate that the “abstracting abstract machines” approach carry over to more high-level definitional interpreters that are not expressed as state machines. In particular, they show that this approach can reconstruct the CFA2 control flow analyses [44, 45] based on push-down structures that also computes precise return information.

Trace partitioning [12, 24, 36] is a technique that can increase the precision of analyses, by exploiting the execution traces of a program to create a partition of its abstract states. We can view the call site paths used as keys in the environments and in the caches of k-CFA as a mean to partition the abstract states: k-CFA computes an environment and a cache for every call site path. We obtained these paths as abstractions of the traces, which is consistent with the trace partitioning approach. Our functions ϕ identify several contexts, and we use them to define context sensitivity policies. A similar approach is followed in [37, Chap. 8.2], where instances—timestamps that are used to differentiatate instances of formal parameters as in the AAM approach—are abstracted using Galois connections.

8 Conclusions and Future Work
We have proposed a substitution-based small-step operational semantics for the λ-calculus, that produces a trace of the control flow events that take place during reduction. We have devised a series of abstractions which, when applied to the trace-based semantics, derive k-CFA by abstract interpretation. Based on the obtained abstract interpretation, we have produced a semantic justification for the constraint-based presentation of k-CFA.

We have also employed abstract interpretation to define the VCFA analysis, a novel control-flow analysis that utilises non-finite domains, and takes advantage of widening to ensure termination of the analyser. We have shown that VCFA can compute more precise results than those provided by k-CFA analyses, especially when numeric domains are involved. Our experiments show that using widening in CFAs is not only possible but also brings benefits.

Using the same approach, we think m-CFA [10, 29] could be obtained using a different abstraction on traces, that would keep frames instead of call sites. The question remains open whether we could also derive the more recent pushdown CFAs [6, 7, 11, 44, 45]. This could be achieved possibly by employing more precise abstractions. For example, we have not exploited so far the order in which events occur, nor have we used the contents of call annotations, that record in the calling contexts which functions were called and on which arguments. This extra information is likely to help us justify the well-bracketed aspect of pushdown analyses.

While the experimental results for VCFA are encouraging concerning precision, further study is necessary to measure how the analysis performs on larger, real-world programs, and how it compares to pushdown-based analyses. To answer these questions, we need to improve the performance of our analyser. A natural target for improvement is to make our top-down solver incremental, so that unnecessary recomputations are avoided. Another improvement would be to use a more expressive abstract domain for closures, that keeps information on function results and would help the widening operator lose less information.

We have based our semantic development on a semantics of traces. This idea could be pushed further, to analyse other intentional properties of programs, such as effect analysis, or complexity analysis. Moreover, we could also enrich the traces with the inputs of the program—an approach similar to the analysis of input-output relations [30]. This would result in a relational control-flow analysis that detects which functions are called at a given program point, depending on the inputs of the program.

References
brown.edu/pub/techreports/92/cs92-25.pdf


